Documentation for `run.py`: Profiling Side-Channel Attacks on ASCON using 3-bit Subkeys

Contents

1 Overview **2**

2 Dataset Structure and Leakage Model **2**
- 2.1 HDF5 Layout . . . 2
- 2.2 3-bit Subkey Extraction . . . 2
- 2.3 Subkey Labels for Profiling . . . 3

3 SNR-Based Window Selection **3**
- 3.1 Motivation . . . 3
- 3.2 Multi-Class SNR Definition . . . 4
- 3.3 Window Selection . . . 4

4 Models and Loss Functions **4**
- 4.1 Models . . . 4
- 4.2 Loss Functions . . . 5

5 Training Procedure for a Single Subkey **5**
- 5.1 Dataset Construction . . . 5
- 5.2 Train/Validation Split . . . 6
- 5.3 Optimization . . . 6

6 Attack Phase and SCA Metrics **6**
- 6.1 Subkey Likelihood Aggregation . . . 6
- 6.2 Guessing Entropy and Success Rate per Subkey . . . 7
- 6.3 Global GE and SR for Each Model . . . 7

7 Key Reconstruction from 3-bit Subkeys **7**
- 7.1 Decoding the 3 Bits . . . 7
- 7.2 Geometry of `select_subkey` . . . 8
- 7.3 Probabilistic Bit-Level Reconstruction . . . 8
- 7.4 Comparison with True Key Bits . . . 9

8 Model Ranking and Output Structure **9**
- 8.1 Ranking Criteria . . . 9
- 8.2 Directory Layout . . . 10

9 Usage Summary **10**

1 Overview

This document describes the design and rationale of the script `run.py`, which implements a profiling side-channel attack on ASCON using multi-class classification of 3-bit subkeys.

The script is designed to:

- Use the `random_keys` group of an HDF5 file for *profiling* (training/validation).
- Use the `fixed_keys` group of the same HDF5 file for the *attack* (key recovery).
- For each subkey index i (called `BitNum` in the code, ranging from 0 to 63), treat the 3-bit value returned by `ascon.select_subkey(i, key)` as an 8-class label.
- Compute a multi-class SNR curve for each subkey and select an informative time window for training.
- Train different neural models (MLP, CNN1D, TCN, TinyTransformer) to classify these subkeys.
- Aggregate predictions over the `fixed_keys` traces to perform subkey recovery, then reconstruct the first 64 bits of the key using all 3 bits per subkey.
- Evaluate models using Guessing Entropy (GE), Success Rate (SR), and key-bit similarity.

The goal is both to perform a realistic key recovery and to compare models for side-channel resilience.

2 Dataset Structure and Leakage Model

2.1 HDF5 Layout

The script assumes an HDF5 file with the following groups:

- `/random_keys`
  - `traces`: matrix $X^{(r)} \in \mathbb{R}^{N_r \times T}$, where N_r is the number of profiling traces and T is the number of time samples per trace.
  - `metadata`: structured array with at least a field `key`, such that `metadata['key']` yields a matrix of shape (N_r, L) with L key bytes in `uint8`.
- `/fixed_keys`
  - `traces`: matrix $X^{(f)} \in \mathbb{R}^{N_f \times T}$, where N_f is the number of attack traces.
  - `metadata`: structured array with the same layout as above, including the `key` field.

In the unprotected case (the one handled directly by the script), `metadata['key']` already contains the full secret key bytes. All traces in the `fixed_keys` group share the same secret key.

2.2 3-bit Subkey Extraction

The script imports helpers from `ascon_helper.py`. In particular, it uses the function

```
ascon.select_subkey(bitnum, key_bytes, array=False)
```

where:

- `bitnum` is an integer in $[0, 63]$,

- `key_bytes` is the key as a sequence of bytes,
- `array=False` returns an integer in $\{0, 1, \ldots, 7\}$.

Internally, `select_subkey` constructs a 3-bit vector $(b_0, b_1, b_2) \in \{0,1\}^3$ derived from specific key-bit positions. For $0 \leq i < 64$, the helper uses parameters `reg_size = 64`, `reg_num_0 = 4`, and a diffusion array `dr` such that:

$$(s_1, s_2) = \texttt{dr[reg\_num\_0]} \in \{0, \ldots, 63\}^2,$$

and then defines:

$$\begin{aligned} b_0(i) &= \text{key-bit at position } i, \\ b_1(i) &= \text{key-bit at position } (i + s_1) \bmod 64, \\ b_2(i) &= \text{key-bit at position } (i + s_2) \bmod 64. \end{aligned}$$

The function `select_subkey(i, key)` then returns the integer

$$s(i) = (b_0(i) \ll 2) \mid (b_1(i) \ll 1) \mid b_2(i) \in \{0, \ldots, 7\}.$$

Thus, for each index $i \in \{0, \ldots, 63\}$, the 3-bit subkey encodes three specific key bits. Importantly, each key bit k_j (for $j \in \{0, \ldots, 63\}$) appears in *three* subkeys:

- as $b_0(j)$,
- as $b_1((j - s_1) \bmod 64)$,
- as $b_2((j - s_2) \bmod 64)$.

This redundancy is exploited later for key reconstruction.

2.3 Subkey Labels for Profiling

The script first extracts all keys from the metadata:

```
combined_keys_random = extract_combined_keys_from_metadata(metadata_random)
combined_keys_fixed = extract_combined_keys_from_metadata(metadata_fixed)
```

where each of these is a matrix of shape (N, L) in `uint8`.

Then, it computes all 3-bit subkeys for all traces and all bit indices:

```
subkeys_random = compute_subkeys_for_all_bitnums(combined_keys_random, num_bitnums=64)
subkeys_fixed = compute_subkeys_for_all_bitnums(combined_keys_fixed, num_bitnums=64)
```

The function `compute_subkeys_for_all_bitnums` fills an array

$$\texttt{subkeys}[i, \texttt{bitnum}] = s(\texttt{bitnum}) \in \{0, \ldots, 7\}$$

for each trace i.

For a fixed `bitnum` (denoted by b), the label vector used for training is:

$$\mathbf{y}_b^{(r)} = \texttt{subkeys\_random}[:, b] \in \{0, \ldots, 7\}^{N_r}.$$

3 SNR-Based Window Selection

3.1 Motivation

Traces typically contain a large number of samples T, while the leakage of interest is localized in certain time intervals (related to particular operations in the ASCON implementation). Training on the full trace can be inefficient and may dilute the discriminative information.

To focus the models on the most informative region, the script computes a multi-class Signal-to-Noise Ratio (SNR) curve over time and selects a window centered on the maximum SNR.

3.2 Multi-Class SNR Definition

Let $X \in \mathbb{R}^{N \times T}$ be the matrix of traces, and $y \in \{0, \dots, C-1\}^N$ the corresponding class labels (here $C = 8$ for the 3-bit subkeys). For each time sample t and class c, define:

- $\mu_c(t)$ = mean of all traces at time t such that $y_i = c$.
- $\sigma_c^2(t)$ = variance of those traces at time t.

Then, the multi-class SNR at time t is:

$$\mathrm{SNR}(t) = \frac{\mathrm{Var}_c\big(\mu_c(t)\big)}{\mathrm{Mean}_c\big(\sigma_c^2(t)\big)} = \frac{\dfrac{1}{C}\displaystyle\sum_{c=0}^{C-1}\big(\mu_c(t) - \bar{\mu}(t)\big)^2}{\dfrac{1}{C}\displaystyle\sum_{c=0}^{C-1}\sigma_c^2(t)},$$

where $\bar{\mu}(t)$ is the mean of $\mu_c(t)$ over c.

High SNR indicates that the means of the classes are well separated relative to their within-class noise, making those time samples more informative for classification.

3.3 Window Selection

For a chosen window size W (argument `-window-size`), the script computes $\mathrm{SNR}(t)$ for all $t \in \{0, \dots, T-1\}$, finds the index $t^\star$ of the maximum SNR, and selects a contiguous window of W samples centered around $t^\star$:

$$\mathrm{start} = \max(0, t^\star - \lfloor W/2 \rfloor), \quad \mathrm{end} = \min(T, \mathrm{start} + W).$$

Boundaries are adjusted to ensure the window length is as close to W as possible.

This window is then applied to both `random_keys` and `fixed_keys` traces for that subkey index.

4 Models and Loss Functions

4.1 Models

The script supports four models, all implemented in PyTorch:

- **MLP** (baseline): defined in the script.
  - Input: flattened window $x \in \mathbb{R}^W$.
  - Several fully-connected layers with ReLU activations and dropout.
  - Output: logits in $\mathbb{R}^8$ (one per subkey class).
- **CNN1D**: imported from `models/cnn.py`.
  - Input: $x \in \mathbb{R}^{1 \times W}$.
  - Stack of Conv1d-BN-ReLU-MaxPool blocks.
  - Global average pooling over time, followed by a small MLP classifier.
  - Output: logits in $\mathbb{R}^8$.
- **TCN**: imported from `models/tcn.py`.
  - CNN "stem" for local feature extraction.

  - Stack of dilated residual blocks (TCN-style).
  - Global average pooling and MLP classifier.

- **TinyTransformer**: imported from `models/transformer.py`.
  - Convolutional stem to project $\mathbb{R}^{1\times W}$ into a sequence of d-dimensional tokens.
  - Depthwise convolution for positional mixing.
  - Stack of Transformer blocks (multi-head self-attention + FFN).
  - Attention pooling over time to obtain a single feature vector.

All models finally output a vector of logits $\ell \in \mathbb{R}^8$ for the 3-bit subkey classification.

4.2 Loss Functions

Two loss functions are available, selectable via `-loss-type`:

- **Cross-Entropy (CE)**:
$$\mathcal{L}_{\text{CE}}(\ell, y) = -\log \frac{\exp(\ell_y)}{\sum_{k=0}^{7} \exp(\ell_k)}.$$

- **RankingLoss (RkL)**: implemented in `loss_functions/rankingloss.py` as pairwise logistic loss:
$$\mathcal{L}_{\text{RkL}}(\ell, y) = \frac{1}{C-1} \sum_{k\neq y} \log\bigl(1 + \exp(-\alpha(\ell_y - \ell_k))\bigr),$$
where $\alpha > 0$ is a scaling hyperparameter.

The CE loss is standard for multi-class classification. The RankingLoss is designed to emphasize ordering of the correct class relative to incorrect ones, which aligns well with Guessing Entropy and ranking-based SCA metrics.

5 Training Procedure for a Single Subkey

5.1 Dataset Construction

For a fixed subkey index $b \in \{0, \dots, 63\}$, the script:

1. Extracts labels for the profiling traces:
$$y_i^{(r)} = \texttt{subkeys\_random}[i, b], \quad i = 0, \dots, N_r - 1,$$
and for the attack traces:
$$y_j^{(f)} = \texttt{subkeys\_fixed}[j, b], \quad j = 0, \dots, N_f - 1.$$
All attack traces share the same subkey value, since all those traces use the same secret key.

2. Computes the SNR curve on $X^{(r)}$ with labels $y^{(r)}$ and selects a window $[t_{\text{start}}, t_{\text{end}})$.

3. Applies the window to both sets:
$$\tilde{X}^{(r)} \in \mathbb{R}^{N_r \times W}, \quad \tilde{X}^{(f)} \in \mathbb{R}^{N_f \times W}.$$

4. Constructs a `SCASubkeyDataset` with normalization (mean and std) learned from $\tilde{X}^{(r)}$ and applied to both profiling and attack sets.

5.2 Train/Validation Split

The profiling dataset is split into training and validation subsets, e.g.:

$$N_{\text{train}} = 0.8 \cdot N_r, \quad N_{\text{val}} = N_r - N_{\text{train}}.$$

The split is random but deterministic given a fixed seed.

5.3 Optimization

For each epoch:

1. **Training phase**:
   - For each batch of (x, y) from the training loader, the model produces logits ℓ.
   - The chosen loss (CE or RankingLoss) is computed and backpropagated.
   - The optimizer (Adam) updates the model parameters.
   - Batch-wise training loss and accuracy (on the subkey label) are accumulated for logging purposes.
2. **Validation phase**:
   - The model is evaluated on the validation set without gradient updates.
   - Validation loss and accuracy are computed and logged.

The script saves two plots per subkey and model:

- Training vs. validation loss across epochs.
- Training vs. validation accuracy across epochs (diagnostic only).

6 Attack Phase and SCA Metrics

6.1 Subkey Likelihood Aggregation

Once a model is trained for subkey index b, it is evaluated on all attack traces (windowed):

1. For each batch of attack traces x, logits $\ell \in \mathbb{R}^{B\times 8}$ are computed.
2. Log-softmax is applied to obtain $\log p(c \mid x)$ for each class $c \in \{0, \ldots, 7\}$.
3. The log-likelihoods are accumulated over all attack traces:

$$\log L(c) = \sum_{j=1}^{N_f} \log p(c \mid x_j^{(f)}).$$

The estimated subkey for index b is:

$$\hat{s}_b = \arg \max_{c\in\{0,\ldots,7\}} \log L(c).$$

In addition to $\hat{s}_b$, the script keeps the full vector $\big(\log L(c)\big)_{c=0}^{7}$ for each subkey index b. These aggregated log-likelihoods are later used for bit-level probabilistic key reconstruction.

6.2 Guessing Entropy and Success Rate per Subkey

Let $s_b^\star$ be the true subkey value for index b (which is the same for all attack traces, since they share the same key).

To compute Guessing Entropy and Success Rate at subkey level, the script:

1. Sorts the classes by decreasing log-likelihood:
$$\pi_b(1), \dots, \pi_b(8)$$
where $\pi_b(1)$ is the most likely subkey value and $\pi_b(8)$ the least likely.

2. Finds the rank r_b such that
$$s_b^\star = \pi_b(r_b).$$
The rank r_b is in $\{1, \dots, 8\}$.

3. Defines the per-subkey success indicator:
$$\mathrm{SR}_b = \begin{cases} 1, & \text{if } r_b = 1, \\ 0, & \text{otherwise.} \end{cases}$$

6.3 Global GE and SR for Each Model

For each model, after processing all 64 subkey indices:

- The *Guessing Entropy* is defined as the average rank of the correct subkey:
$$\mathrm{GE} = \frac{1}{64} \sum_{b=0}^{63} r_b.$$
A lower GE is better (1 is perfect, 4.5 is approximately random for 8 classes).

- The *Success Rate* is defined as the fraction of subkeys for which the model places the correct subkey at rank 1:
$$\mathrm{SR} = \frac{1}{64} \sum_{b=0}^{63} \mathrm{SR}_b.$$

These two metrics are standard in SCA and provide a more meaningful measure of attack performance than simple classification accuracy.

7 Key Reconstruction from 3-bit Subkeys

7.1 Decoding the 3 Bits

The script stores all recovered subkeys in an array $\hat{s}_b \in \{0, \dots, 7\}$ for $b = 0, \dots, 63$. For each b, it decodes:

$$\begin{aligned} \hat{b}_0(b) &= (\hat{s}_b \gg 2) \,\&\, 1, \\ \hat{b}_1(b) &= (\hat{s}_b \gg 1) \,\&\, 1, \\ \hat{b}_2(b) &= \hat{s}_b \,\&\, 1, \end{aligned}$$

where `&` is the bitwise AND.

Thus, one obtains three bit sequences $\hat{b}_0, \hat{b}_1, \hat{b}_2 \in \{0, 1\}^{64}$.

7.2 Geometry of `select_subkey`

As summarized earlier, the ASCON helper defines, for $i \in \{0, \dots, 63\}$:

$$\begin{aligned} b_0(i) &= k_i, \\ b_1(i) &= k_{(i+s_1) \bmod 64}, \\ b_2(i) &= k_{(i+s_2) \bmod 64}, \end{aligned}$$

where $(s_1, s_2) =$ `ascon.dr[ascon.reg_num_0]` and k_i denotes the i-th bit of the 64-bit key portion.

Conversely, each key bit k_j appears in three subkeys:

- as $b_0(j)$,
- as $b_1((j - s_1) \bmod 64)$,
- as $b_2((j - s_2) \bmod 64)$.

7.3 Probabilistic Bit-Level Reconstruction

While a simple majority vote over the three occurrences of each key bit is possible, the script implements a more informative *probabilistic* reconstruction that combines the full subkey log-likelihoods.

Recall that for each subkey index b the attack phase produces $\log L_b(c)$ for all $c \in \{0, \dots, 7\}$, where $L_b(c)$ is the aggregated likelihood (up to a multiplicative constant) for the hypothesis that the 3-bit subkey at index b equals c.

Each class c can be decoded into three bits $(b_0, b_1, b_2) \in \{0, 1\}^3$ via

$$b_0 = (c \gg 2) \,\&\, 1, \quad b_1 = (c \gg 1) \,\&\, 1, \quad b_2 = c \,\&\, 1.$$

For a fixed key bit index $j \in \{0, \dots, 63\}$, the helper `select_subkey` implies that k_j appears in exactly three subkeys:

- as bit b_0 in subkey index $i_0 = j$,
- as bit b_1 in subkey index $i_1 = (j - s_1) \bmod 64$,
- as bit b_2 in subkey index $i_2 = (j - s_2) \bmod 64$.

For each hypothesis $k_j = b$ with $b \in \{0, 1\}$, the script proceeds as follows:

1. For each occurrence (i, r) of k_j, with $r \in \{0, 1, 2\}$ denoting the role b_r, consider the subset of subkey classes

$$\Omega_{i,r}(b) = \{c \in \{0, \dots, 7\} \mid \text{the } r\text{-th bit of class } c \text{ equals } b\}.$$

2. Compute a log-likelihood contribution for that occurrence by *log-sum-exp* over the compatible classes:

$$\log L_{i,r}(k_j = b) \approx \log \sum_{c \in \Omega_{i,r}(b)} \exp\bigl(\log L_i(c)\bigr).$$

3. Approximate the total log-likelihood for $k_j = b$ as the sum of the three contributions:

$$\log L(k_j = b) \approx \log L_{i_0,0}(k_j = b) + \log L_{i_1,1}(k_j = b) + \log L_{i_2,2}(k_j = b).$$

Finally, the reconstructed bit is chosen as the more likely hypothesis:

$$\hat{k}_j = \arg\max_{b\in\{0,1\}} \log L(k_j = b).$$

This procedure uses the full distribution over the 8 subkey classes at each index, rather than only the most likely class. The three occurrences of each key bit are thus combined in a "soft" manner, where highly confident subkeys contribute more strongly to the final decision than uncertain ones. The result is a recovered 64-bit vector $\hat{\boldsymbol{k}} = (\hat{k}_0, \ldots, \hat{k}_{63})$.

7.4 Comparison with True Key Bits

The script extracts the true key from the metadata of the `fixed_keys` group:

```
true_key_bytes_fixed = combined_keys_fixed[0]
true_key_bits64 = key_bytes_to_bits_first64(true_key_bytes_fixed)
```

where `key_bytes_to_bits_first64` converts the first 8 bytes of the key to a 64-bit big-endian vector.

Then, for each model, it compares the reconstructed bits $\hat{\boldsymbol{k}}$ with $\boldsymbol{k}^\star$:

- Number of correct bits:
$$N_{\text{correct}} = \sum_{j=0}^{63} \mathbf{1}\{\hat{k}_j = k_j^\star\}.$$
- Key-bit similarity:
$$\text{Sim} = \frac{N_{\text{correct}}}{64}.$$
- Hamming distance:
$$d_H = 64 - N_{\text{correct}}.$$

These values are reported per model and saved in `summary.txt` and a plot `true_vs_recovered.png`.

8 Model Ranking and Output Structure

8.1 Ranking Criteria

For each model, the script records:

- GE: average rank of the correct subkey across 64 subkeys.
- SR: fraction of subkeys where the correct subkey is ranked first.
- Key-bit similarity and Hamming distance.

The final ranking is based primarily on SCA metrics:

1. Higher SR is better.
2. For equal SR, lower GE is better.
3. For equal SR and GE, higher key-bit similarity is better.

8.2 Directory Layout

Under the base output directory `-output-dir`, the script creates:

- `global_ranking.txt`: global ranking of the models with SR, GE, similarity and Hamming distance.
- For each model (e.g. `model_MLP`, `model_CNN`, etc.):
  - `summary.txt`: GE, SR, similarity, recovered subkeys, true and recovered key bits.
  - `true_vs_recovered.png`: plot of true vs recovered bits for indices 0–63.
  - One subdirectory per subkey index:
    - `subkey_XX/snr.png`: multi-class SNR curve and selected window center.
    - `subkey_XX/train_val_loss.png`: training and validation loss curves.
    - `subkey_XX/train_val_acc.png`: training and validation accuracy curves (diagnostic only).

9 Usage Summary

The script is invoked from the command line, for example:

```
python run.py \
  --h5-path /path/to/ascon_unprotected.h5 \
  --output-dir ./results_ascon \
  --loss-type ce \
  --models mlp cnn tcn transformer \
  --epochs 50 \
  --batch-size 256 \
  --window-size 500
```

Key arguments:

- `-h5-path`: path to the HDF5 file containing `random_keys` and `fixed_keys`.
- `-output-dir`: base directory for all results.
- `-loss-type`: `ce` or `ranking`.
- `-models`: subset of {`mlp`,`cnn`,`tcn`, `transformer`}.
- `-epochs`, `-batch-size`, `-window-size`: standard training hyperparameters.
- `-device`: `cpu`, `cuda` or `xpu`. If not set, the script automatically prefers `cuda` if available, otherwise `xpu` if available, and falls back to `cpu`.

With this pipeline, each model is evaluated using SCA-relevant metrics (GE, SR, key-bit similarity) in a fully automated key-recovery experiment on ASCON traces.